LA-UR 97-3890

Approved for public release;
distribution is unlimited

Title:

Compressed Fields in the POOMA Framework: Design,
Implementation, and Performance

Author(s):

Federico Bassetti
Steve Karmesin
Tim Williams
John Reynders

RECEIVED

DEC 16 1997

OSTI

Submitted to:

For external distribution as requested.

MASTER

DISTRIBUTION OF THIS DOCUMENT IS UNLIMITED

Los Alamos
National Laboratory

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the University of California for the U.S. Department of Energy under contract W-7405-ENG-36. By acceptance of this article, the publisher recognizes that the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. The Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.

DISCLAIMER

**Portions of this document may be illegible
in electronic image products. Images are
produced from the best available original
document.**

Compressed Fields in the POOMA Framework: Design, Implementation, and Performance

Federico Bassetti, Steve Karmesin, Tim Williams, John Reynders

Advanced Computing Laboratory
Los Alamos National Laboratory
CIC/ACL MS B287
Los Alamos, NM 87545
{fede,karmesin,zippy,reynders}@acl.lanl.gov

Abstract. The POOMA framework[8, 4] is a C++ class library for doing large scale parallel computations. POOMA's Field class implements a multidimensional data parallel array annotated with mesh coordinate information. The Field class breaks up the whole domain into sub-blocks and transparently compresses constant blocks down to a single value to improve the memory and computational efficiency. This allows the user to simply and efficiently simulate phenomena that are taking place in a small fraction of the domain. We examine the cache and computational benefits associated with this capability.

The algorithm implementation is based on the fact that if we have a large constant region in the field we only need to store one value for that region. In this paper we will describe the design of compressed fields, their implementation in the POOMA framework, and the performance of this technique on an Silicon Graphics Inc. Origin 2000.

1 Introduction

POOMA[8, 4], Parallel Object-Oriented Methods and Applications, is an object-oriented framework for a class of scientific computing applications that require the memory and computation speed afforded only by existing and future high-performance computers. We use an object-oriented approach to manage the complexity of explicit parallel programming by encapsulating the data distribution and communication among processors under a mostly data-parallel interface.

POOMA is implemented as a set of templated C++ class libraries. It evolved from, and is driven by, the needs of its finite set of applications that currently span structured mesh partial differential equations and particle in cell (PIC) calculations.

1.1 General Motivation: Multimaterial Hydrodynamics

One of the applications of the POOMA framework is the solution of multimaterial hydrodynamics problems[9]. The requirements of this application led into the development and implementation of field compression in the POOMA

framework. These compression techniques apply well not only to multimaterial hydrodynamics but also to other applications with similar characteristics.

This multimaterial hydrodynamics problem requires solving a coupled system of fluid and material-property equations for compressible flow of multiple material species. The general approach is to solve discretized fluid dynamical equations on a mesh and use numerical implementations of physics-based models for the material properties. Properties include equations of state and elasticity/plasticity; these enter into the fluid equations as parameters. The numerical method is explicit finite differencing, with a Lagrangian phase half-timestep followed by a remap to the Eulerian grid.

The calculations are expected to be on the order of 100 million grid cells with on the order of 10 materials with widely varying properties (both gases and solids, for example). These materials are each present in a subset of the domain, and the locations of those subsets shifts over the course of the simulation.

For a fluid simulation such as this, key abstractions from the problem domain include the computational mesh and fluid fields (material-dependent and material-independent) discretized on the mesh. Because not all materials exist in all grid cells, and because of the enormous memory demands of simulations, it is essential to store fluid field quantities and material properties in grid cells only for those materials that exist there.

1.2 Expected Benefits of Compression

- **Total Storage.** Memory is one of the most valuable resources, and is the limiting factor in this application. By compressing constant regions we can often drastically reduce the required storage for a given computation.
- **Computation.** Compression enables to get the same results while using fewer instructions. Applying compression to a field the computation that normally would be done on every element needs to be done only once per compressed region.
- **Memory Bandwidth.** Because less data is stored and used, compression allows us to reduce the number of cache transfers needed by the computation. This can reduce the time required for a computation even in cases where the number of operations remains the same.

2 Design and Implementation

2.1 POOMA Fields and Virtual Nodes

The POOMA Field class[8] is the primary class that represents the abstraction of a fluid field discretized on a mesh. The Field class is parameterized on the type of its elements (such as double, int, vector, tensor), the dimensionality, the mesh type and the centering on the mesh.

One can express finite differences on Field quantities using array-indexing operations like $(F[I+1][J][K] + F[I-1][J][K])/2.0$. Here F is a Field object, $\{I,J,K\}$ are Index

objects that represent strided integer ranges. This expression represents a stencil of computation on the range $\{I,J,K\}$ evaluated using expression templates[7]. We have designed differential operator functions which will act on Field objects and encapsulate the actual finite differencing.

The Field data is broken up into subdomains, each being a discretized contiguous sub-domain of the entire computational domain, which we call virtual nodes or vnodes. In a sequential context like the one under consideration here, the vnodes are created and assigned to the same processor, and in a parallel context different vnodes would be allocated on different processors.

2.2 Inside Compression

Compression as implemented in POOMA satisfies three main constraints: it reduces the required storage, it preserves the high level interface of a data-parallel array, and it is straightforward to implement. It does this by examining each of the vnodes for each Field, and if the value is constant it compresses it down to a single value. The Field can then be used as if it were full, but in regions where the value is constant it acts more efficiently.

The system rarely needs to actually examine every element of a vnode to determine if it can be compressed. When assigning to a Field it can examine the compression structure of the right hand side to determine how the left hand will be compressed. In cases where the system cannot determine the possible compression in this manner, the user can force a Field to examine every element and compress wherever possible.

This form of compression is granular: each sub-domain is either compressed or not. Another form of compression would involve allocating only those individual cells that need to be allocated, but that would make the representation essentially unstructured. We would then have to store lists of pointers to neighbors resulting in a different form of inefficiency. In certain applications this will be more efficient than the granular approach used here, and we are investigating this area for future applications.

3 Performance Evaluation

The experiments we ran to demonstrate the effects of field compression were on the SGI Origin 2000[2], a cache-coherent non-uniform memory access multiprocessor. The Origin system consist of up to 512 nodes interconnected by a scalable Craylink network. Each node consists of one or two R10000 processors, and up to 4GB of memory. The analysis presented in the paper is based on results obtained using a single processor of the Origin system.

The performance data gathered in the experiments made use of the built-in performance monitor interfaces [10,5] of the R10000. A set of 32 different counters are available, which allow collection of data for the execution of the whole code or for just a portion of the code. Using these counters we measured number of cycles executed, number of instructions executed, number of floating

point instructions executed, primary and secondary cache misses, load and store instructions executed, and others. Combining the different counters with each other we managed to compute different types of metrics[6, 3, 1] that we used to characterize the performance of our testcodes.

(a) Execution time: before reaching the steady state the compressed version is significantly faster. The lines represent the times spent in each step. Two different vnode numbers are used: 16 and 64. The flat lines are from uncompressed runs, and the stepped ones are from runs with compression. The 64 vnode case approaches more gradually (more steps) the steady state than the 16 vnode one.

(b) Memory usage: before reaching the steady state the compressed version demand significantly less memory. The lines represent the percentage of uncompressed memory limit used by the code at every step. When the whole field is uncompressed the value for each step is 1, values smaller than 1 reflect the use of some compression. The timestep at which the 128 vnode case demands the whole capacity is delayed by 250 steps compared to the 16 vnode case.

Fig. 1. The effect of compression timestep by timestep in a 2D diffusion.

3.1 2D diffusion testcode

In Figure 1 we characterize the effects of compression on a 2D diffusion code. The code is a simple 2D time-dependent diffusion with unit diffusion coefficients, and is chosen to be simple to make the effects of compression clear.

In the initial state the heat field is zero everywhere except for one grid cell in the center of the domain. The computation at every step is a 9-point stencil

with constant coefficients. In Figure 1 a field of 1000x1000 elements is used to characterize the performance when the number of vnodes varies. We show how the field compression reduces the execution time and memory requirements.

3.2 Moving area testcode

In Figure 2 we characterize the effects of compression using a testcode that models a region of a field which moves its center timestep after timestep. In the testcode, we model a 2D field, a region made of just one point, and it moves on a straight line from one side of the field to the other.

Objective of this testcode is to show how a code could benefit from compression, if the computation is such that compression can take place even after the initialization of the field (as in the 2D diffusion). Also, with this testcode is used to show the overhead induced by forcing a compression during the computation.

(a) Compressed fraction: the amount of data that can be compressed is dependent on the number of vnodes for a given problem size. Compression is maintained at the same percentage, in almost all the iterations. The dropping points are showing the effect of crossing vnodes. The moving area has point on more than one vnodes. More vnodes allow to have, on average, higher compression.

(b) Cost of Compression: try to compress a field has an impact on performance. Performance improves when the number call to the compress routine decreases. A call to ensure max compression is needed only when the region of computation has moved to a different vnode. The result show are obtained for a field of 512x512 elements, with 256 vnodes.

Fig. 2. The effect of compression timestep by timestep in a code that models a field with a moving area.

4 Measurements

In Figure 1 we show the effect of compressed fields in the 2D diffusion code. In Figure 1.a we show how compression speeds up the execution time. In the first approximately 400 time steps in the 64 vnode case and in the approximately 250 steps in the 16 vnode case the field is non-zero only near the center of the domain and the outer parts of the grid are compressed. The uncompressed runs were performed for comparison by setting the initial value of the field to non-constant values everywhere. We see that when compression is possible the execution time is lower, and we see the jumps in execution time as the outer areas of the grid become uncompressed. A larger number of vnodes provides a finer granularity and potentially better compression, but if the whole field is uncompressed there is more overhead associated with doing the computation.

Figure 1.b shows how the demand for memory changes when compression is used. In cases when compression is disabled by using a non-constant initial condition, the memory demanded is equal to the problem size, and is allocated from the very first step. When compression is possible only a fraction of that memory is actually used. This fraction is small in the initial steps and grows until the steady state is reached with the same memory requirement as the uncompressed case. These jumps in the memory requirements occur in the same time steps as the jumps in the execution time.

In Figure 2.a we present the impact of compression when we vary the granularity. One can see that with higher number of vnodes, more compression is possible on average. Also, this testcode enables us to observe how the compressed percentage varies when the center of the computing region moves. Notice that, if compression is not applied frequently during the computation the percentage of compression in the field will be constantly decreasing, and in a similar way performance too (as shown in 1).

In Figure 2.b instead shows the impact of invoking the compress function. One can see that try to compress at every timestep is not too efficient, because no further compression is possible if the computation covers one or more vnodes. So the ideal way of using compression is to recognize when the computation has moved to other vnodes and compress those that have been uncompressed by the computation and that are now unused.

5 Conclusions

In this study we introduced the design and implementation of compression techniques in the POOMA framework. The compression is hidden in the implementation of the Field class and is completely transparent to the user, and uses less memory and execution time when sub-regions of a field are constant.

In this paper we presented the performance of uniprocessor runs of the testcode making use of the compression techniques. These same effects will be still present in a distributed memory multiprocessor context.

References

1. S. Goedecker and A. Hoisie. Achieving high performance in numerical computations on risc workstations and parallel systems. June 1997. Mannheim, Germany.
2. James Laudon and Daniel Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *International Symposium on Computer Architecture*, June 1997. Denver, CO.
3. O.Lubeck, Y. Luo, H. Wasserman, and F. Bassetti. Performance evaluation of the sgi origin2000: A memory-centric characterization of lanl ascii applications. Submitted to Supercomputing97.
4. J.V.W. Reynders et al. *POOMA: A Framework for Scientific Simulations on Parallel Architectures*, volume Parallel Programming using C++ by Gregory V. Wilson and Paul Lu, chapter 16, pages 553-594. MIT Press, 1996.
5. Silicon Graphics Inc. *Speedshop User's Guide*. <http://www.sgi.com/Technology/TechPubs/>.
6. Silicon Graphics Inc. *Performance Tuning for the Origin 2000*, 1997. <http://www.sgi.com/Technology/TechPubs/manuals/3000/007-3511-001/html/O2000Tuning.0.html>.
7. Todd Veldhuizen. Expression templates. Technical Report 5, C++ Report 7, June 1995.
8. Tim Williams, John Reynders, and William Humphrey. POOMA User Guide. Advanced Computing Laboratory - Los Alamos National Laboratory, April 1997. <http://www.acl.lanl.gov/pooma/doc/userguide>.
9. Tim J. Williams et al. Multimaterial hydrodynamics application using the pooma framework. submitted to ISCOPE'97.
10. Marco Zagha et al. Performance Analysis Using the MIPS R10000 Performance Counters. In *Supercomputing96*, 1996. <http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/INDEX.HTM>.